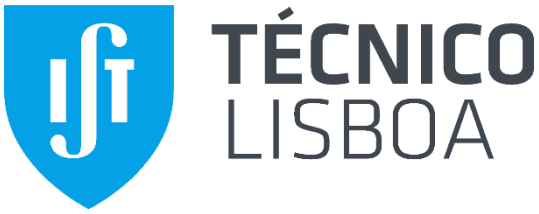

Projeto de Algoritmos e Estruturas de Dados

2º Semestre 20/21

MEEC - Grupo 8

96252 – João Coelho
joaopcoelho@tecnico.ulisboa.pt

97349 – João Nabais

joao.m.nabais@tecnico.ulisboa.pt

Índice

1. Problema Apresentado

O problema apresentado, referido adiante como "AEDMaps", pede, sucintamente, a construção de um algoritmo que consiga navegar entre duas localizações (vértices) dum mapa (obtido como grafo) com diversas restrições.

2. Arquitetura do programa

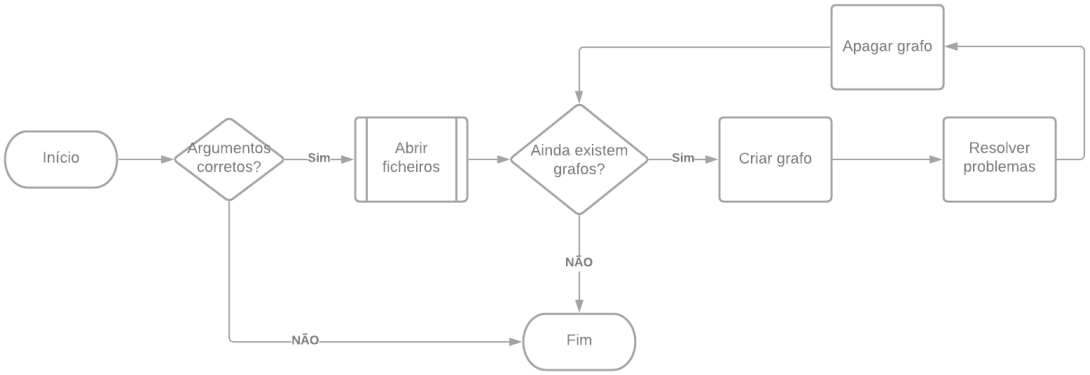

Figura 1 - main

A figura 1 representa a estrutura básica do programa. O processo predefinido "Abrir ficheiros" apenas os abre e coloca os problemas numa lista, de seguida inicia-se o ciclo que corre todos os problemas para os mapas dados.

2.1 **Estruturas de Dados**

No nosso projeto foi implementada uma solução com um vetor em que cada posição é um vértice do grafo e cada um tem uma árvore binária auto-balanceada (neste caso uma AVL) para guardar os vértices a que este está ligado, ou seja, as arestas. Esta escolha foi feita durante a primeira fase de submissões de modo a diminuir o tempo de acesso a um vértice específico em comparação a uma lista de adjacências, passando O(N) para O(logN).

Estrutura data **–** Estrutura principal onde são armazenadas as informações do grafo. Contém dois inteiros, o número de vértices e o número de arestas e um *pointer* do tipo trunk.

Estrutura trunk **–** Estrutura que representa cada nó do grafo onde é armazenado um inteiro, o número de ligações, um *pointer* para *char* para guardar os pontos de interesse (referidos adiante como id's) e um pointer para node para guardar o início (raiz) da árvore de ligações.

Estrutura node **–** Estrutura que representa uma ligação entre dois vértices (aresta). Contém dois inteiros, o "nome" do vértice ligado e a altura na árvore, um *double* para guardar o custo da ligação e dois *pointers* do tipo node que apontam para o nó à esquerda e à direita deste na árvore.

Para guardar os problemas foi criada uma lista simplesmente ligada na qual cada nó contém os valores referentes ao respetivo problema.

Estrutura probs **–** Contém três inteiros, dois guardam o "nome" de dois vértices e o outro guarda o valor de k (utilizado em C1 e D1), uma *string* que guarda o tipo de problema (A1, B1, C1 ou D1), um caracter (utilizado em B1), um double (utilizado em B1) e um *pointer* para probs.

Na fase final de entrega foi necessário recorrer à implementação do algoritmo de Dijkstra e, para tal, implementámos uma *priority queue*.

Estrutura PQueue **–** Esta estrutura contém um *pointer* para QData, dois inteiros para guardar o número de elementos presentes na queue num dado instante e o número total de elementos e um *int pointer* para criar um vetor auxiliar ao funcionamento.

Estrutura QData **–** Estrutura auxiliar para guardar os dados necessários de cada vértice. Contém uma variável do tipo *bool* para verificar se foi visitado ou não, dois inteiros de modo a guardar o seu índice na *queue* e o "nome" do vértice através do qual passou antes de ele próprio e duas variáveis do tipo *double* para guardar o custo da aresta e o custo do caminho.

Estrutura parentArray **–** Contém dois inteiros para guardar em cada posição o "pai" do vértice daquela posição e o custo da aresta até ao mesmo.

Na primeira fase de entrega foi necessário recorrer a um *Breadth-first search*, assim foi necessário a utilização de uma *queue*. Para tal foram usadas as duas seguintes estruturas:

Estrutura queue **–** Estrutura constituída por dois pointers do tipo qnode para o topo e fundo da *queue*.

Estrutura qnode **–** Estrutura que contém um *pointer* para qnode e dois inteiros, o "nome" do vértice e o número de ligações passadas para lá chegar.

2.2 Leitura e Seleção de Problemas

A leitura dos problemas para a respetiva lista é feita pela função *extractProbs* que têm o seguinte fluxograma:

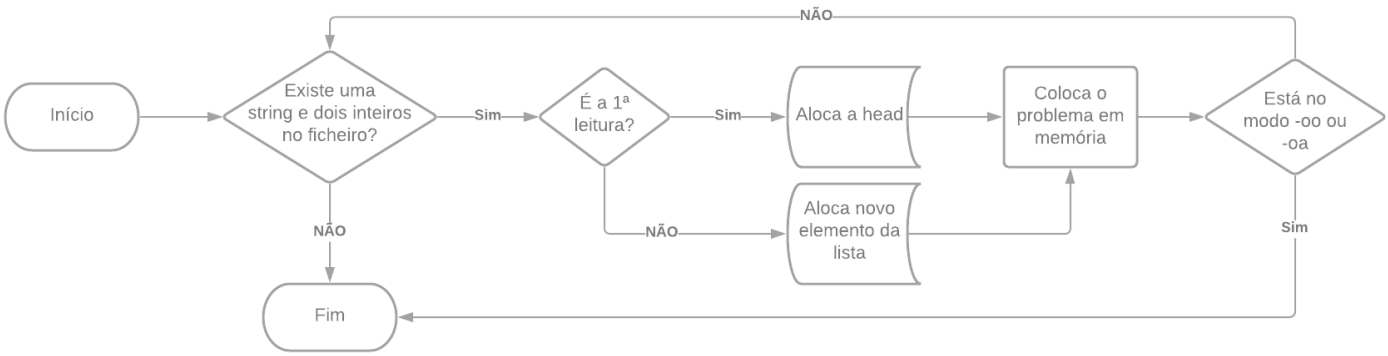

Figura 2 - extractProbs

A seleção dos problemas é feita na função *selectProblems*, para a qual se envia a lista de problemas, o grafo e o *pointer* do ficheiro de saída.

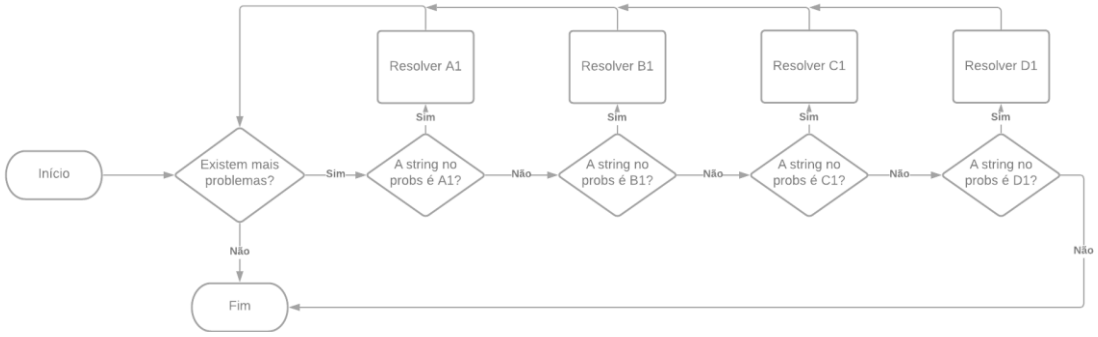

Figura 3 - selectProblems

2.3 **Resolução dos problemas**

Para a resolução dos quatro problemas apresentados recorremos à implementação do algoritmo de Dijkstra, ou seja, um *shortest path algorithm*. Na nossa implementação utilizamos um *priority queue* de forma similar a uma *heap* de modo a diminuir os tempos de acesso à mesma de O(N) para O(log N).

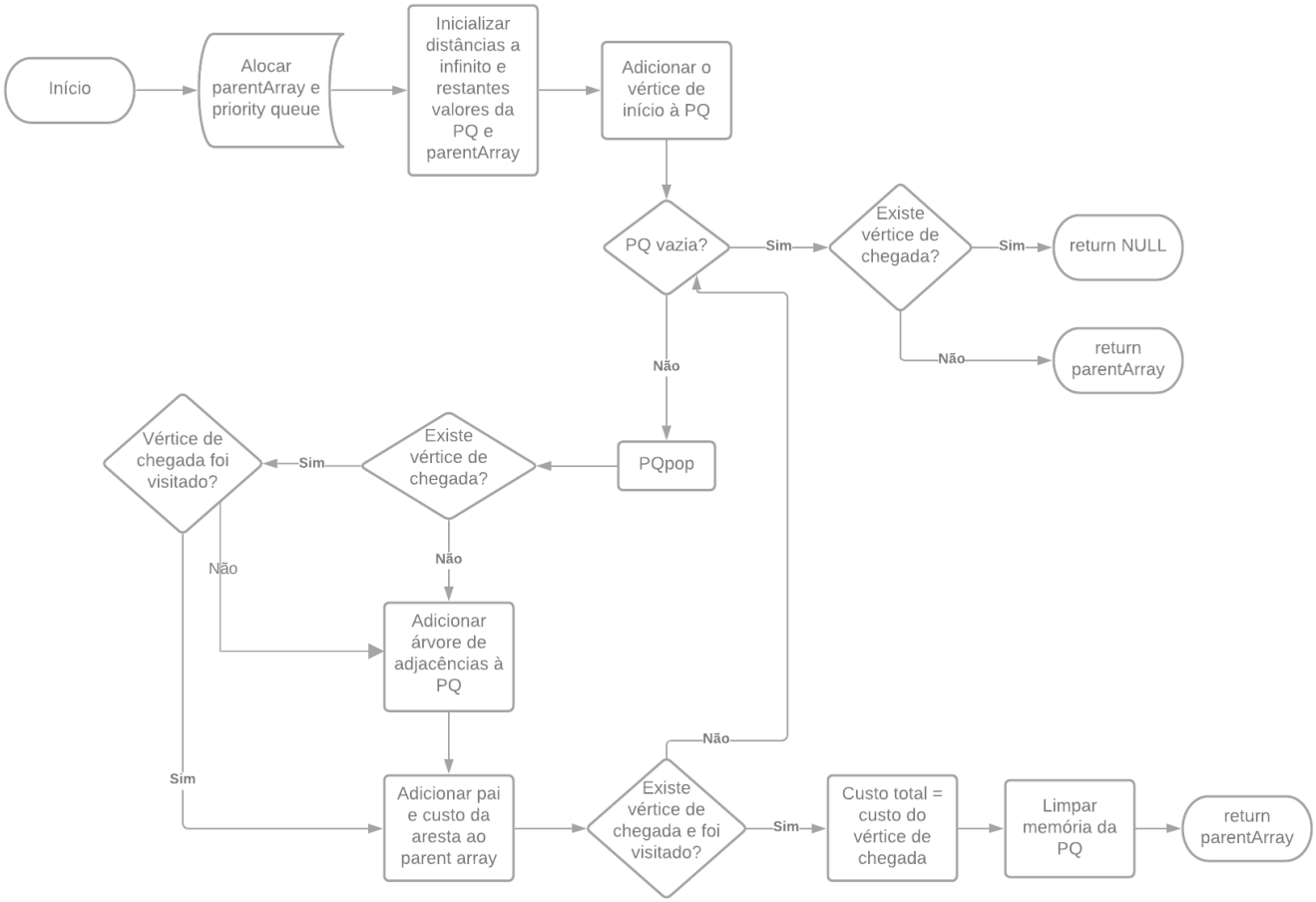

Figura 4 - dijkstra

O modo de funcionamento A1 pede o caminho mais barato entre dois vértices, o que indica, uma única utilização do dijkstra. A função pode ser descrita pelo seguinte fluxograma:

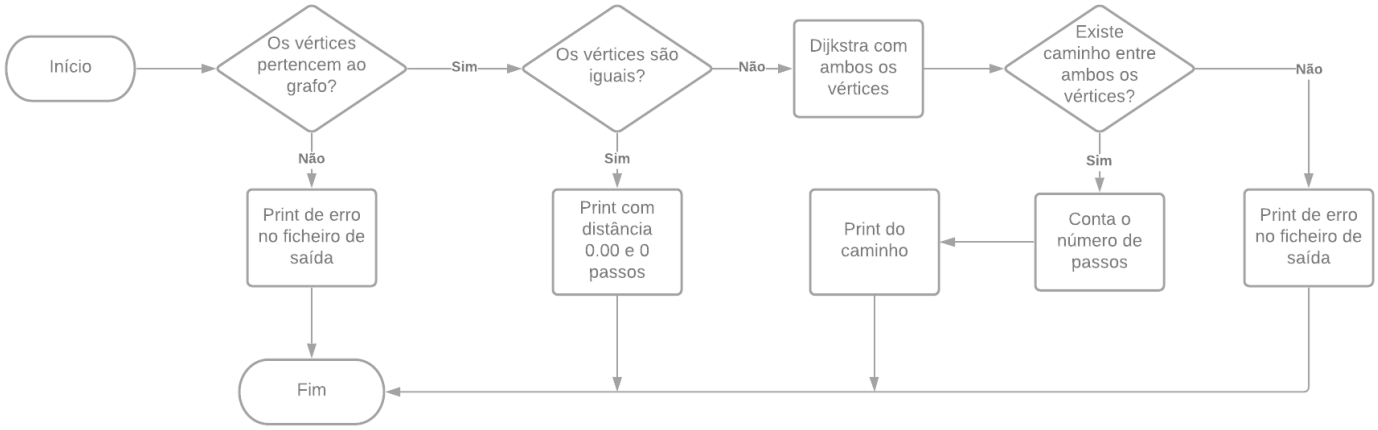

Figura 5 - A1

O modo B1 pede a determinação do caminho mais barato entre dois vértices passando por um vértice com um ponto de interesse. Para tal a nossa solução corre o algoritmo de Dijkstra duas vezes na sua forma integral, ou seja, sem vértice de destino e de seguida determina os custos totais entre o vértice de partida, os vértices com aquele ponto de interesse e o vértice final e guarda o vértice para o qual o custo é menor.

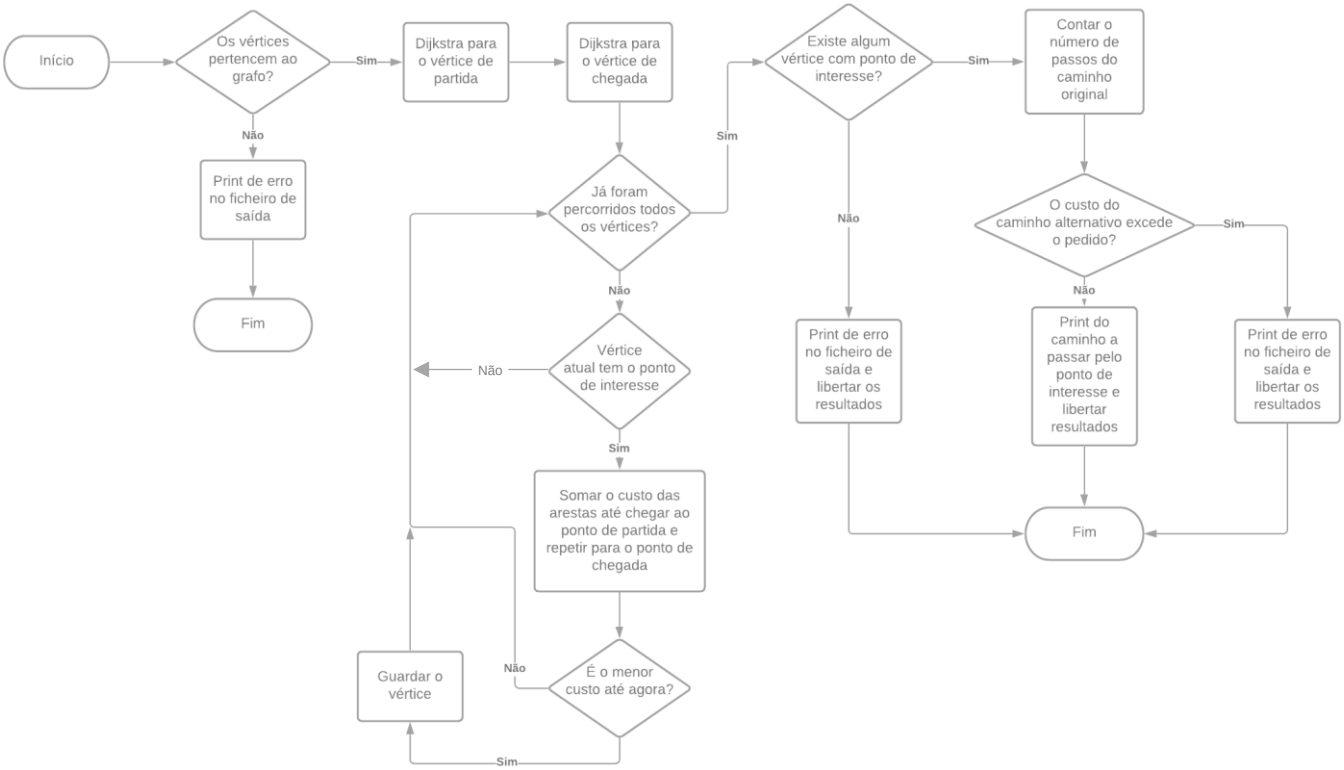

Figura 6 - B1

Os modos C1 e D1 são bastante similares sendo em C1 pedido o menor caminho entre dois pontos tendo em conta que uma localização está interdita (vértice impedido) e em D1 é pedido o menor caminho entre dois vértices tendo em conta que não se pode utilizar uma estrada (aresta impedida). Em ambos os casos a forma de impedir que o vértice ou a aresta seja visto como uma opção válida ao correr o algoritmo de Dijkstra é uma verificação do vértice (e o seu "pai" no caso de D1) que está a ser adicionado na função onde os vértices são adicionados à *priority queue*.

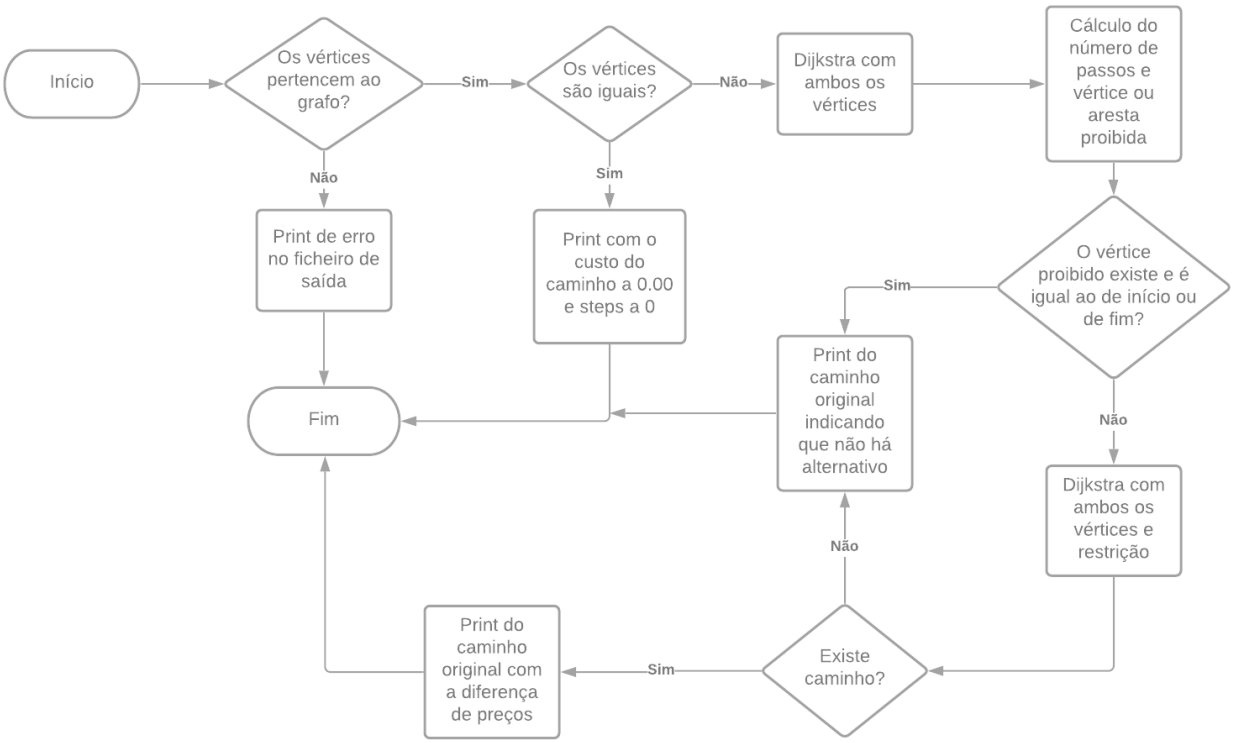

Figura 7 - C1 & D1

3. Análise de requisitos computacionais

Na nossa implementação do projeto foram utilizadas árvores auto-balanceadas de modo a diminuir o tempo de procura de uma ligação em comparação ao tempo de procura de uma ligação em listas de adjacência. Esta escolha foi feita com base no que foi pedido na primeira parte do projeto o que requeria a implementação de um BFS. Na utilização do mesmo era necessário a busca de ligações entre dois vértices e o uso de árvores em comparação a listas diminuía o tempo de processamento de O(N) para O(log N) sendo N o número de ligações de um dado vértice. Em questão à memória utilizada para guardar os vértices tem-se O(V+2E) visto que foi guardado em memória um vetor de V vértices sendo em cada vértice guardada as suas ligações diretas logo 2E no total, sendo E o número de arestas. Para desalocar o grafo fornecido é necessário desalocar V árvores e visto que a complexidade temporal de desalocar a árvore é de O(logV) então temos que desalocar o grafo tem uma complexidade temporal de O(VlogV).

Quanto às implementações das soluções dos problemas temos primeiro que analisar a complexidade temporal do algoritmo de Dijkstra implementado para realizar a procura em

cada problema. No algoritmo em questão o tempo de inicialização da *priority queue* é O(N), o tempo de extração do valor mínimo é O(VlogV) e o tempo de realizar a operação *decrease key*, ou seja, diminuir o valor (custo) de um determinado vértice e rearranjar a *priority queue* de modo a manter a condição de acervo é de O(ElogV), logo tem-se que a complexidade final do algoritmo de Dijkstra será O(ElogV). Assim tem-se que a complexidade temporal de A1 é igual à do Dijkstra, ou seja, O(ElogV). De forma similar o modo C1 e D1 também têm complexidade O(ElogV) visto que correm o algoritmo de Dijkstra duas vezes. Quanto ao modo B1 temos que este corre o algoritmo de Dijkstra duas vezes logo O(ElogV) e de seguida percorre o vetor de vértices e verifica os seus identificadores logo O(V) o que indica que o modo B1 possui também complexidade temporal O(ElogV).

4. Exemplo de funcionamento

Ao evocar o programa através da linha de comandos é primeiro verificado se o programa foi chamado com número de argumentos corretos. De seguida é aberto o ficheiro de problemas e estes são extraídos e colocados numa lista simplesmente ligada do tipo probs. Após esta extração entramos num ciclo que corre todos os mapas, verificando se existem dois inteiros restantes no ficheiro (número de vértices e de arestas) enviando-os para a função *createGraph* onde, como o nome indica, é criado o grafo.

Nesta função é alocado um vetor do tipo trunk com V (número de vértices) posições, onde, no seguinte ciclo, são extraídos do ficheiro as *strings* que indicam os pontos de interesse de cada vértice e é alocada memória para as guardar na estrutura. No ciclo seguinte são extraídas as arestas e o custo das mesmas que são inseridas na árvore de adjacência.

Após a criação do grafo a lista de problemas é enviada em conjunto com o grafo para a função *selectProblems* onde é identificado cada problema e resolvido como descrito em 2.3. Depois da resolução do problema é utilizada a função *CleanMem* na qual é desalocada toda a memória alocada para o grafo.

O ciclo de criação, resolução e desalocação repete-se desde que nos encontremos num dos modos que lê mais do que um mapa e ainda existam mapas, após o qual é desalocada a lista de problemas e são fechados os ficheiros de mapas e de saída.

5. Bibliografia

Dijkstra's algorithm – Wikipedia

https://en.wikipedia.org/wiki/Dijkstra's_algorithm

Dijkstra's shortest path algorithm

https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

Sedgewick, R. (2001). *Algorithms in c* : Addison-Wesley Professional.